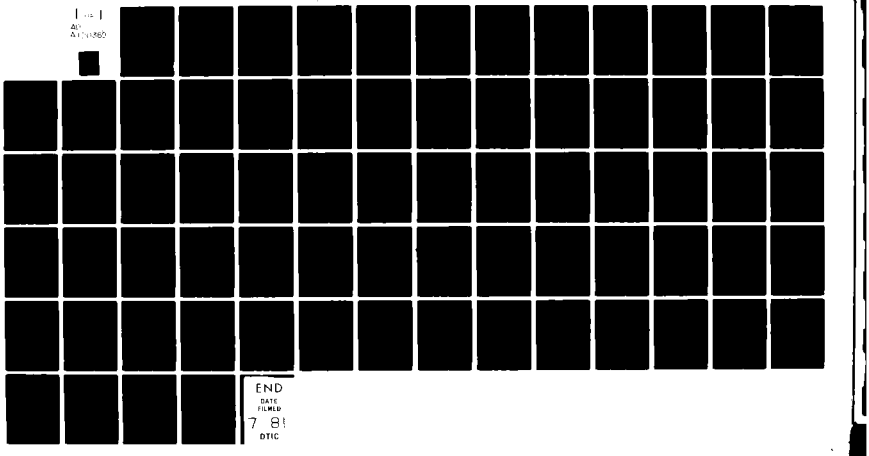
AD-A100 369 OFFICE OF THE UNDER SECRETARY OF DEFENSE FOR RESEARCH--ETC F/G 9/2
DEPARTMENT OF DEFENSE REQUIREMENTS FOR HIGH ORDER COMPUTER PROG--ETC(U)
JUN 76

UNCLASSIFIED NL



END
DATE FILMED
7 81
DTIC

DEPARTMENT OF DEFENSE

REQUIREMENTS FOR HIGH ORDER COMPUTER PROGRAMMING LANGUAGES.

"TINMAN".

A

JUNE 1976

OFFICE OF THE UNDER SECRETARY OF DEFENSE
WASHINGTON, D.C. 20301

RESEARCH AND
ENGINEERING

Dear Friend of Ada:

Thank you for your interest in Ada.

Your name has been added to the Ada mailing list, and occasionally you will receive information from the Ada Joint Program Office concerning the status of the Ada program.

Under the Freedom of Information Act, the Ada Joint Program Office (AJPO) mailing list is being made available on the USC-ECLB computer. If you object to inclusion of your name on this public list, please inform the AJPO in writing. To help keep the list up-to-date, please notify the AJPO of address changes.

Sincerely,

for Larry E. Druffel, Lt. Col., USAF
Director, Ada Joint Program Office

This "Tinman" document represents the requirements of the Department of Defense for High Order Computer Programming Languages. This is the position of the representatives of the Military Departments, as presently formulated; however, it should be regarded as a living document to be modified as requirements become more specific, as technical capabilities are better appreciated, or as expansion of the explainations becomes indicated. Comments are, therefore, actively solicited on all levels and may be addressed to the Chairman of the High Order Language Working Group:

Lt. Col. William A. Whitaker
Defense Advanced Research Projects Agency
1400 Wilson Boulvard
Arlington, Virginia 22209

A

And the Lord said, Behold, the people is one, and they have all one language; and this they begin to do: and now nothing will be restrained from them, which they have imagined to do.

Genesis XI 6

I. Introduction

Like most large computer users, the United States Department of Defense has been plagued with a proliferation of high order languages and incompatible systems serving the "same" language. The DoD's problems are, in principal, no different from the rest of the computer user community; they are simply larger as is the use of computers. Further, DoD systems are often individually very large and very long lived. Because of the intimate integration of the computer resources with the rest of a large defense system, it has not been possible to change computer subsystems with frequency which might characterize a commercial operation. As a result, maintenance of the computer system is both long term and dynamic, and maintenance in many cases involves modification of the system to respond to new threats. Defense systems are often composed of interacting but independently developed subsystems, sometimes brought into existence over a period of years, all of which must be served by a common but evolving hardware base. In such an environment, the Department of Defense finds itself spending an increasingly larger fraction of its systems resources on software. High order language commonality and the resulting flexibility would provide a powerful tool for reducing the high cost of software in the DoD.

With each of the Military Departments studying this problem and making proposals for common languages, it was clear that the greatest benefit could be reaped by providing languages common across the Department of Defense. In January 1975, a DoD High Order Lanugage Working Group (HOLWG) was chartered by DDR&E with representatives from the Military Departments to investigate the requirements and specifications for programming language commonality, to compare these with existing approaches, and to recommend adoption or implementation of the necessary common language or languages. Until the matter is resolved, the DoD will not support any further implementation of new high order languages in R&D programs.

The first task of the HOL Working Group was to formulate a set of requirements consistent with the levies of the Military Departments. This effort proceeded as follows:

While there is no intent to replace the already standard COBOL and FORTRAN in their field of application, initially there was no reason to unnecessarily restrict the requirements to only those areas such as real-time applications or weapons systems where the major problems are, although this area must receive emphasis. Therefore, requirements were solicited from as broad a base as possible, to be prioritized later as required. Further, inquiries were not restricted to those programs presently using high order languages, rather, a major thrust of the effort is to provide HOL's to meet the requirements of those who are now constrained to use an assembly language for lack of a suitable HOL. — next page

A major problem was to formulate the inquiry in such a way that meaningful requirements would result. On the one hand, one wishes to avoid broad generalities such as the obvious goals of efficiency, ease of coding, readability, etc., which, while are very real goals, are insufficiently quantified to provide any detailed guidance. On the other hand, to over specify the requirements at the level of individual language features would merely formalize past idiosyncrasies and magnify distinctions which do not represent fundamental differences. It has proved to be impossible to define rigorously the exact level requirement desired; and, therefore, a "strawman" of preliminary requirements was established to define this level by illustration. The "strawman" was not intended to be complete or consistent, rather it was deliberately provocative in order to elicit the widest possible comment. It was forwarded to the Military Departments and, by them, to their various operating organizations. In addition, it was distributed to other government agencies, to the academic community, and to industry, through industry organizations and military contractors, and by direct inquiry. A number of individuals outside the U.S. were also solicited for comments. Hundreds of individuals and organizations have had an opportunity to examine this document and provide inputs. the bulk of such inputs were positive and useful. Negative comments were almost universally based on a misunderstanding of the status of the document. Believing this to purport to be a complete and consistent set of requirements some correctly pointed out that it was no such thing.

The results of four months of such input were put together in a more concrete form, one which could then pretend to represent a fairly complete set of requirements, although still a tentative set. This document was called the"woodenman, " and it too was distributed widely. It provided a more rigorous framework for specific comment. On the basis of all inputs and the official responses from each of the Military Departments, a more complete set of requirements has evolved and is presented here as a "tinman." This document represents a set of requirements for high order computer programming language consistent with the input from the Military Services.

II. Definition

A high order language (HOL) is one which provides compression of a computer program such that one HOL statement represents many machine language instructions.

Experienced programmers in the early 50's found that they would generate similar instruction sequences often enough that it proved useful to either set aside duplicated blocks of cards to be inserted in the program as required, or in a more automated fashion, to develop macros which might be defined once to the assembly program and thereafter invoked by shorthand notation. From such beginnings high order languages were evolved as a method for speeding up programming. The extension and development of this technique led to the more powerful concept of a higher level of programming language.

The first truly successful high order language was FORTRAN. It was developed for the IBM-704 and in the original version most of its constructs were identical to those commonly employed by machine language programmers for that machine; in some sense, it was inspired by this specific machine. But in addition, the constructs were sufficiently general that it could be adapted fairly easily to other machines and the practical reality emerged the first time of constructing a program which was machine independent, even across manufacturers. So powerful was this concept at the time that in the early 60's the use of such high order languages was heralded as automatic programming.(Annual Review of Automatic Programming, Pergamon Press, Oxford, 1960, 1961, 1963, 1964, etc.)

In 1959 a meeting was held in the Pentagon to establish a common business-oriented language, the outcome of which was COBOL (Programming Languages: History and Fundamentals, Jean E. Sammet, Prentice-Hall, Edgewood Cliffs, N.J. 1969). Other similar attempts at fairly general languages such as ALGOL, as well as those designed for a more specialized users such as CMS-2 or JOVIAL have found their somewhat smaller niche in the marketplace.

Even higher level specialized languages have come into existence such as GPSS or SIMSCRIPT for simulation programs, or ATLAS for automatic test equipment. These are problem-specific applications packages which may be generated through the use of one of the other high order language programming languages and are not in the range of the present effort.

By high order language, this report will mean a computer programming language of the general level of COBOL or FORTRAN primarily designed for programmers to communicate with computers. It is non-problem specific and should be able to exercise almost all of the capability normally used by those programming in machine language and at a reduction of an order of magnitude in the number of

statements necessary. The language is relatively machine independent, representing possible logical constructs rather than specific hardware function, and is translated by a compiler into machine code to be run on a target machine.

III. General Goals

In this section are listed the general goals of a DoD software program. Some of the ways in which these impact on the detailed desired language characteristics set out in the next section are illustrated.

These goals, and the point of view they indicate, are generated by the chalenges of the Defense Systems Community which is concerned with what may be loosely termed Embedded Computer Resources. They have a fair overlap with, but a different emphisis from, those of the Automatic Data Processing Community.

There is no real attempt to order or weigh these goals at this point nor is there any claim that the list is complete. It is merely a very brief summary of discussions taking place during the evolution of the requirements.

COST

The overriding goal of all technology and standardization programs are to reduce the cost of Defense systems. Software is an area in which there are great opportunities in cost savings which can be brought about through actions of the DoD. Computer hardware savings are much more difficult to generate. Since the Department of Defense represents only a very small fraction of the computer hardware market, it must effectively content itself with the off-shoots of the mainstream of computer technology. On the other hand, the DoD is the main customer for software; and the major support for basic computer science and technology. It is, therefore in a position to influence the state-of-the-art and to materially change the software development process. A number of managerial initiatives to this end are presently under development. Many of them, including the setting up of libraries for software exchange, building up of a tool environment for improved software production facilities, distribution of government furnished tools, encouragement of the use of high order languages for maintenance ease, etc., depend upon, or will be facilitated by, restriction to a minimal number of common high order languages.

Much of each new software system is repetitive of processes that have been written before. A few new ideas may be added, linkages may be changed, but the bulk of the work is not new. Yet, most systems are started from scratch with little dependence on previous work other than what the individual programmers might remember. There is little high order language commonality. A new language or dialect, or at least an incompatible compiler is often generated for each project and it is impossible to sustain the creation of a truly powerful production environment.

Facilities are often locked into one particular computer because of the great expense of transferring software to the new machine. There are several proposals and on-going work in the DoD concerning the facilities for the translation of software just in a commercial-like environment. This problem and its implications may be far more serious in some of our more specialized applications. A fully supported and controlled common language facility would eliminate most of these difficulties and provide more flexibility to the managers to computer resources.

RESPONSIVENESS AND TIMELINESS

By its very nature, software is plagued with underestimates. It represents a major fraction of the total systems capability, but is physically insignificant. There are no power requirements, no drop tests, no critical materials consumed. It is basically unsubstantive, at best, a few bits on a magnetic tape. It is by nature easy and quick to change and, by implication, it is difficult to imagine how it can be lengthy to produce. While it is easy to change, it is sometimes very hard to make it correct, as opposed to just different. As a result, estimates of the time to complete a project are often overly optimistic, sometimes so much so that the hardware is ready before the software. The cost of such delays is not just the extra man hours to bring the software into shape, but must be computed on the loss of the entire system during the period of delay. The technology then does not exist for accurately estimating software schedules other than by experience and analogy. Language can only contribute to this in that it is a part of a stable software production environment.

An actual reduction in the time to produce software would be possible if the language reduced the coding time of the project. The mere existence of the high order language normally reduces this coding time by a fair amount and certain language features may act to improve this. One must remember, however, that the coding portion of a large project normally represents only ten or fifteen percent of the total time involved and the promotion of writing ease must, therefore, be weighted with other portions of the program production cycle. A major fraction of this cycle, sometimes up to half, is involved with debugging and checking and, therefore, the thrust to more reliable code must be stronger.

Specification and design is a similarly large fraction of the effort and factors such as modularity which simplify the design are also important. The overriding consideration must, however, be the possibility of eliminating most of the effort altogether through reusing portions of both code and design which have been done before. The ability to draw upon large amounts of past work, either through libraries, previous experience, or automatic programming tools, offer the greatest possibility for payoff in timeliness and responsiveness.

It must be pointed out that this thrust towards transportability emphasizes language transportability because the elements to be transported are individual subprograms and routines, not necessarily systems concepts or coupling schemea. It is the lack of acceptance of the possibility of using previously designed and certified piece parts which separates present day hardware and software methods.

RELIABILITY

Perhaps the most characteristically military requirement for DoD software is reliability. While all gradations of this exist in the DoD, certainly the combination of extreme complexity in systems, some almost untestable, and the life and death implications, not just of individuals but of millions, must be unique. This reliability requirement is reflected in verification, validation, and certification during the production process, as well as back-up in alternate systems, safeguards, etc. A significant portion of the software cost is tied up in this thrust, although often very subtly. Language characteristics which promote the production of reliable code must be weighted very highly.

Commonality allowing the reuse of previously certified blocks of code is also of importance. Unfortunately, the technology does not exist to automatically prove the correctness of code at a usable level. Should it eventually be possible to prove that a code will do exactly what it is intended to do and also, as in the security question, will not do anything that is not intended, the language features which make this possible would have very high priority in the DoD. Until then, perhaps the best we can do is make available facilities for investigation and experiments in this direction.

Other features promoting reliability in the DoD programming environment, would be simplicity and unambiguity of the language and readability of the constructs. It has been suggested that certain new programming techniques inherently produce better and more reliable code. A language should make it possible to use such techniques, although the brunt of enforcement must be on the programming management environment.

MAINTAINABILITY

A major portion of Defense cost in software systems occurs during the maintenance phase. These systems are often very long lived, which is to be expected from their size and complexity and interaction with other systems. They must be kept viable for upwards of twenty years during which time major changes can take place in the threat, the employment concept, and even portions of the hardware. As a result, maintenance is often a blanket for continued development and improvement.

At the present time the statistics are somewhat deceptive in this area. On the basis of perhaps inadequate data, one would estimate that twice as much is now being spent on development as on maintenance. However, this figure is badly distorted in that the largest systems are still under development, while those in maintenance were those produced in much less ambitious times. The inventory is still growing rapidly; when it stabalizes,one might expect to see a reversal of the ratio. In most individual systems, the Operations and Maintenance costs exceed those for Development and Acquisition by several times.

Over the maintenance period, the responsibility for the program will pass through many hands both individually and corporately, between contractor and government organizations. Over time and with each transition, knowledge is lost. The more readable and definitive the program, the better knowledge will be transmitted. Transportability of the main program is usually not a question in this phase, but transportability in machine independence of the tools necessary to maintain and support that program are vital as the facilities used in this maintenance will change much more rapidly than will the main system hardware. Likewise, one of the main tools is the personnel involved. Reduction of the total number of languages with which one must work with will increase the efficiency and the reliability of the maintenance effort. Documentation of all sorts is most important in this phase. Whatever characteristics of the language that promote or support documentation, internal or external, will contribute to maintainability. A certain amount of self-documentation results just from the use of a high order language. Modularity and structured programming can further aid. Modern techniques in this area must be available.

TRAINING

Training is a difficult factor to quantify. Formal courses are only a very small fraction of the training that a programmer gets. The bulk of it is in the form of on-job-training or simply experience. In many cases, for instance, scientific programming, the training is woefully inadequate or even non-existent. The disparity in the productivity between programmers may, in many cases, be traced to inadequate training. Better than the productivity measure would be the cost resulting from poorly written programs. Reduction of the number of languages in use would simplify the training burden.

It might be argued, particularly by expert programmers, that transitioning to a new language is fairly simple and should only take a few days of study and practice. They would then argue against paying any penalty. Unfortunately, a census of the active programmers on DoD projects, whether in-house or contract, would uncover only a small minority of such versatile experts. The average programmer is often tied to a single language and, at that, not completely proficient in its techniques and tools.

The training burden would also argue for a simple language with a minimum of unambiguous constructs. Ideally, the constructs would fail soft so that ignorance of a facility would simply deny that facility and not result in unexpected side affects.

TRANSPORTABILITY

One of the two main technical advantages of high order languages in general is transportability, sometimes termed "machine independence." The advantage of being able to write a program in sufficiently general form that it could be compiled on to more than one computer is appealing for a number of reasons. One can transfer capabilities developed in one project to another working on different equipment, and one can expect greater lifetime for software when it can be carried on from one generation of machine to the next in the same installation. Both of these advantages are clear cut and everyone with a few years of software experience has its own examples to cite in this regard.

The language requirements associated with transportability depend on the situation. In the very early days of SHARE, the only really large-scale computer was the 704 and a great deal of interchange took place for programs written with fairly standard interfaces in machine language. This was entirely adequate for that time. The present diversity of machines requires a different technique. The use of high order language and compiling down to machine code makes a certain amount of machine independence possible but not necessarily inevitable. Any good high order language should lend itself to efficient compilers for most any machine. With appropriate linking mechanisms, programs from various languages could be put together as required. Unfortunately, the practical world intervenes and the production of good compilers for every language and every machine is not a reasonable expectation, nor is sufficient familiarity on the part of majority of programmers with a large number of languages. To make the overall software environment relatively machine independent therefore requires a minimal number of high order languages with reasonably general constructs and active support for wide availability of tools and compilers. This would further suggest that high order language compilers written in that high order language would be most appropriate.

One of the snares that has appeared in the past inhibiting transportability has been the occurrence of incompatible implementations of a single high order language. A language might be insufficiently or ambiguously specified such that different implementers making decisions in an uncontrolled fashion produce incompatible implementations. Further, each one may have his own ideas for additions to the defined language or perhaps may invoke subsets leaving out more difficult to implement or momentarily unnecessary features. We have examples of literally dozens of different implementations of a single language. These are the cruelest mockery of language commonality. Differences are sometimes so subtle as to be difficult to find and lead to undiscovered catastrophes. Such experiences give programmers an almost paranoid skepticism of claims of transportability. To make transportability a reality will, therefore, require not only a rigid and unambiguous language specification, but control or denial of subsets and supersets and rigorous certification of all implementations. It will further require the widespread availablity of such certified compilers for many machines.

The payoff from transportability could be enormous. Many programming chores are simply redoing things which have been done before. Each project usually only adds a small fraction to the sum total of knowledge. If it could draw upon past work, cost, timeliness and, reliability, would all be markedly improved. So far from reality is this today that the techniques involving libraries, etc., have not been worked out. A common high order language is simply one step, albeit an important one, towards this goal.

READABILITY/WRITABILITY

The other major advantage initially claimed for high order languages was that they were easier to read and write, being closer to natural language than is machine code. Claims in this direction were sometimes carried to an extreme and some were promoted so highly that their advocates would have you believe that the coder need not know anything about the machine, simply write out the requirements in English. History does not always support these claims. Many such languages advocated on this basis now are the ones who have the most specialized programmers as their users. The DoD environment is sufficiently large and specialized that the allocation of personnel specifically for software is the rule anyway and there is no strong requirement to eliminate other specialists in this area. This is in contradistinction to a very small commercial firm which may have insufficient demands to justify a full-time programming staff.

Our requirement for readability and writability is therefore primarily among specialists. The desire to communicate with a computer system in natural language is an entirely different one in the DoD and involves query languages or applications programs outside the scope of this high order language effort.

Readability is clearly more important to the DoD than writability. The program is written once, but may have to be read dozens of times over over a period of years for verification, modification, etc. This is certainly true all weapons systems applications and even most of our scientific and simulation programs are very long lived. This requirement is very much different from one which might be generated, for instance, in a university environment when a program may have a life of only a few months and a single person working on it.

Ease of writing is not an inconsiderable goal but one which may be promoted through intelligent terminals, preprocessors, interactive systems, etc. The language evaluation should favor readability where conflict arises.

EFFICIENCY

Efficiency is probably the one overriding consideration on which high order languages stand or fall. Whether it be in runtime speed or memory requirements, these quantifiable costs of using a high order language in preference to machine language or assembly code is almost universally cited as reason for rejecting HOL in a project. Often a detailed study of the matter is not accomplished, so strong do feelings run on this subject. Many changes have taken in place in the technology in recent years so as to reduce the importance of this matter, but it is still such an emotion-charged issue that it must be recognized as the principle customer concern.

Much of the pressure that existed some years ago towards efficiency is no longer relevant. In former times the cost of memory was such a large fraction of the total system expenditure that it might seem reasonable to sacrifice many other advantages in order to obtain memory efficiency. This can no longer be said to be the case. Software, particularly the maintenance portion, is becoming much more important than the hardware. Hardware costs are falling at such a rate that the change in technology over just a few months can decrease the effective cost of memory enough to offset any loss in efficiency, while the cost of going to machine code might result in a much longer delay in developing the program. Likewise, in speed there is often a penalty to pay which is invariably judged so high, often on the basis of experience with bad language implementations, that is common folklore that "HOL cannot be used in real-time applications" or that one must at least have the capability of dropping down into machine code for "time-critical" portions of the system. Yet the cost of increasing the central processor hardware capability by some appropriate amount may be far less than the software burden incurred.

While the cost arguments for HOL are valid, they do not take into account some of the realities of system procurements. It has been common in the past that the hardware configuration is determined very early in the process and the software must be made to fit. Under these conditions, particularly when life cycle cost is not an effective control, there is little choice for the project manager but to be driven to efficiency. A similar situation may arise in the later phases of system life when a new threat or other additional burden must be accommodated within existing hardware.

What measures of efficiency are reasonable? Normally, one thinks in terms of a few tens of percent. An implementation that is within ten percent of assembly language code in speed of execution or memory occupied would be difficult to distinguish, the differences in the quality of code produced by even very good programmers is probably greater than this. (In comparisons one only cites very good programmers. Many current implementations may be as good as an average programmer. Further, one must measure against code which is written in a production environment to be maintainable, etc., not using extraordinarily obscure

techniques. Unique machine dependencies may still require machine code inserts.) A thirty percent penalty might be appropriate in some situations and justified in a sufficiently detailed examination of all costs. A fifty percent degradation for going to high order language may be unacceptable for a project in which the efficiency of the project code was of significance. These figures may be overly severe from a strictly statistical view considering that the logic of a code may admit to programmer variations much greater than these, and large improvements in speed efficiency speed must be sought there. Nevertheless, there is no justification for adding on an additional excessive penalty, particularly when the high order language can be very efficient. To illustrate that an HOL can be efficient, take as an extreme example the technique of defining the program logic in a high order language which is then compiled by hand, a technique which has proved useful in the early phases of some projects. Presumably, the compilation in such a case should be very good. There are even now examples of benchmarks for which HOL compilations are better than assembly code. These may be spurious cases, but in the end this is to be expected theoretically since the HOL route gives a more practical vehicle for large-scale optimization.

One of the main difficulties with many existing high order languages is that their implementations have often been poor, particularly languages generated for single projects and languages implemented in a short time with limited resources. It is not surprising that the implementation is not as powerful as that which could be obtained if 10 or 100 times more effort could be devoted to its development and to the tools and techniques for its use. Such powerful implementations are very good and optimize better than the average programmer. Assuming a very minimal number of well-supported languages for the Department of Defense, one could devote sufficient effort to provide the tools and compilers to produce very efficient object code. It is not clear that any other language requirement mitigates against this expectation.

This particular requirement, or solution, would reinforce the thrust to commonality by making it attractive to reduce not only the number of languages supported but a number of different compiler approaches and would call for the type of control of the language which would promote large efforts for tools and compilers. A contributing technological trend is for DoD programs, particularly those which are the most critical real-time programs, is to have available very large computers on which to build the system, even though it may eventually run on a small computer. The large systems can support cross compilers which are much more powerful than those self-hosted on a smaller target machine. While this is not yet a universal procedure, it certainly provides a further opportunity to bring to bear tools to promote efficient programs. In such a case, the requirement on the high order language is to give as much information to the compiler as possible without overly restricting its options. This argument is contrary to the current practice of providing small subsets of a high order language for mini-computers. Such a subset would

only restrict the amount of information available to a compiler and would, of necessity, produce a less efficient code. The computer might produce even poorer code if forced to host on a small-target machine.

ACCEPTABILITY

No language is going to be of value unless it is used. There are many tangible and intangible influences which will determine its final acceptability. The large scale acceptability of compilers is certainly the most influential tangible asset. Perhaps the only factor that this influences in language design should be the ability to write compilers in the high order language.

That the language resembles something the customer is already familiar with is certainly going to be of considerable importance. Arbitrary decisions on formats must be take in the direction of existing DoD practice. The possibility of making translators from existing programming languages to common languages will probably never appear specifically as a requirement, since there is no intent at this time to perform such translations. However, it would be difficult to deny the attractiveness of such a possibility if it existed.

IV. Needed Characteristics

The set of characteristics prescribed below represent a synthesis of the requirements submitted by the Military Departments and are intended to be consistent with the general goals of Section III, to be self-consistent, and to be achievable with existing computer software and hardware technology. The needed characteristics are the requirements to be satisfied by an existing, modified or new language which is selected as a Common HOL. They prescribe capabilities and properties which a common DoD language should possess but are not intended to impose any particular language features or mechanization of those capabilities. The header of each item gives a general description of the needed language characteristic while the subsequent paragraph(s) of its body provide clarification, discuss some of the implications and problems, provide the rationale behind its inclusion, and/or further detail the requirement. The entire text and not just the headers constitute the requirements.

The large number of characteristics reflects an attempt at thoroughness in dealing with the relevant issues. Similarly, the length of the discussion for many items reflects the need to resolve the ambiguities, examine the implications, and demonstrate the feasibility of the compendious statement introducing that characteristic. Because the characteristics address issues in the design, implementation, and use of the language and properties of the resulting product, there should be no correlation between the number of characteristics discussed here and the number of features in a language which satisifes these characteristics. Many of the characteristics will influence the choice of many features, and every feature will be influenced by many of the needed characteristics. Good language design is a unification process. Any language which satisfies these characteristics must be smaller and simpler than the set of issues underlying its choice.

The material reported in the last three sections (K,L,M) was generated by the Services at the same time as the technical characteristics, but is concerned with translators, support software, documentation, training, standards, application libraries, management policy, and procurement practices for the common language and its use. These issues are important. While mistakes and oversights in the technical characteristics can guarantee failure of the common language effort, success is not guaranteed no matter how technically meritorious the resulting language. Success can only be guaranteed by close attention to a variety of administrative issues, including those considered below.

Several of these issues, including those of implementation, documentation, and support will either directly or indirectly affect the acceptability of candidate languages. As with the needed technical characteristics for the common language, the issues raised here are often not resolved at the most detailed level. Until more detailed characteristics of the language come into focus there is no rationale with which to resolve all these issues in detail.

A. DATA AND TYPES
1. Typed Language
2. Data Types
3. Precision
4. Fixed Point Numbers
5. Character Data
6. Arrays
7. Records

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

By the type of a data object is meant the set of objects themselves, the essential properties of those objects and the set of operations which give access to and take advantage of those properties. The author of any correct program in any programming language must, of course, know the types of all data and variables used in his programs. If the program is to be maintainable, modifiable and comprehensible by someone other than its author, the the types of variables, operations, and expressions should be easily determined from the source program. Type specifications in programs provide the redundancy necessary to verify automatically that the programmer has adhered to his own type conventions. Static type definitions also provide information at compile time necessary for production of efficient object code. Compile time determination of types does not preclude the inclusion of language structures for dynamic discrimination among alternative record formats (see A7) or among components of a union type (see E6). Where the subtype or record structure cannot be determined until run time, it should still be fully discriminated in the program text so that all the type checks can be completed at compile time.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

These are the common data types and type generators of most programming languages and object machines. They are sufficient, when used with a data

definition facility (see E6, D6, and J1), to efficiently mechanize other desired types such as complex or vector.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

This is a specification of what the program needs, not what the hardware provides. Machine independence, in the use of approximate value numbers (usually with floating point representation), can be achieved only if the user can place constraints on the translator and object machine without forcing a specific mechanization of the arithmetic. Precision specifications, as the minimum supported by the object code, provide all the power and guarantees needed by the programmer without unnecessarily constraining the object machine realization. Precision specifications will not change the type of reals nor the set of applicable operations. Precison specifications apply to arithmetic operations as well as to the data and therefore should be specified once for a designated scope. This permits different precisions to be used in different parts of a program. Specification of the precision will also contribute to the legibility and implementability of programs.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

Scaled integers are useful approximations to real numbers when dealing with exact quantity fractional values, when the object machine does not have floating point hardware, and when greater precision is required than is available with the floating point hardware. Integers will also be treated as exact quantities with a step size equal to one.

A5. Character sets will be treated as any other enumeration type.

Like any other data type defined by enumeration (see E6), it should be possible to specify the program literal and order of characters. These properties of the character set would be unalterable at run time. The definition of a character set should reflect on the manner it is used within a program and not necessarily on the

print representation a particular physical device associates with a bit pattern at run time. In general, unless all devices use the same character code, run-time translation between character sets will be required. Widely used character sets, such as, USASCII and EBCDIC will be available in a standard library. Note that access to a linear array filled with the characters of an alphabet, A, and indexed by an alphabet, B, will convert strings of characters from B to A.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

This is general enough to permit both arrays which can be allocated at compile or load time and arrays which can be allocated at scope entry, but does not permit dynamic change to the size of constructed arrays. It is sufficient to permit allocation of space pools which the user can manage for allocation of more complex data structures including dynamic arrays. The range of subscript values for any given dimension will be a contiguous subsequence of values from an enumeration type (including integers). The preferable lower bound on the subscript range will be the initial element of an enumeration type or zero, because it often contributes to program efficiency and clarity.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

This provides all that is safe to use in CMS-2 and JOVIAL OVERLAY and in FORTRAN EQUIVALENCE. It permits hierarchically structured data of heterogeneous type, permits records to have alternative structures as long as each structure is fixed at compile time and the choice is fully discriminated at run time, but it does not permit arbitrary references to memory nor the dropping of type checking when handling overlayed structures. The discrimination condition will not be restricted to a field of the record but should be any expression.

B. OPERATIONS
1. Assignment and Reference
2. Equivalence
3. Relationals
4. Arithmetic Operations
5. Truncation and Rounding
6. Boolean Operations
7. Scalar Operations
8. Type Conversion
9. Changes in Numeric Representation
10. I/O Operations
11. Power Set Operations

B1. Assignment and reference operation will be automatically defined for all data types which do *not* manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

The user will be able to declare variables for all data types. Variables are useful only when there are corresponding access and assignment operations. The user will be permitted to define assignment and access operations as part of *encapsulated type definitions* (see E5). Otherwise, they will be automatically defined for types which do not manage the storage for their data. (See D6 for further discussion).

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

Equivalence is an essential universal operation which should not be subject to restriction on its use. There are many useful equivalence operations for some types and a language definition cannot foresee all these for user defined types. Equivalence meaning logical identity and not bit-by-bit comparison on the internal data representation, however, is required for all data types. Proper semantic interpretation of identity requires that equality and identity be the same for atomic data (i.e., numbers, characters, Boolean values, and types defined by enumeration) and that elements of a disjoint types never be identical. Consequently, its usefulness at run time is restricted to data of the same type or to types with nonempty intersections. For floating point numbers identity will be defined as the same within the specified (minimum) precision.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.

Numbers and types defined by enumeration have an obvious ordering which should be available through relational operations. All six relational operations will be included. It will be possible to inhibit ordering definitions when unordered sets are intended.

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

These are the most widely used numeric operations and are available as hardware operations in most machines. Floating point operations will be precise to at least the specified precision.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

These requirements seem obvious, particularly for floating point numbers and yet many of our existing languages truncate the most significant mantissa digits in some mixed and floating point operations.

B6. The built-in Boolean operations will include "and," "or," "not," and "nor." The operations "and" and "or" on scalars will be evaluated in short circuit mode.

Short circuit mode as used here is a semantic rather than an implementation distinction and means that "and" and "or" are in fact control operations which do not evaluate side effects of their second argument if the value of the first argument is "false" or "true," respectively. Short circuit evaluation has no disadvantages over the corresponding computational operations, sometimes produces faster executing code in languages where the user can rely on the short circuit execution, and improves the clarity and maintainability of programs by permitting expressions such

as, "i<=7 & A[i] >x" which could be erroneous were short circuit execution not intended. Note that the equivalence and nonequivalence operations (see B2) are the same as logical equivalence and exclusive-or respectively.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

Conformability will require exactly the same number of components (although a scalar can be considered compatible with any array) and one for one compatibility in type. Correspondence will be by position in similarly shaped arrays. In many situations component by component operations are done on array elements. In fact, a primary reason for having arrays is to permit large numbers of similarly treated objects to have a uniform notation. Operations on data aggregates available directly in the source language hide the details of the sequencing and thereby, simplify the program and make more optimizations available. In addition, they permit simultaneous execution on machines with parallel processing hardware. Although component by component operations will be available for built-in composite data structures which are used to define application-oriented structures, that capability will not be automatically inherited by defined data structures. A matrix might be defined using an array, but it will not inherit the array operations automatically. Multiplication for matrices would, for example, be unnatural, confusing and inconvenient if the product operator for matrices were interpreted as a component by component operation instead of cross product of corresponding row and column vectors. Component by component operations also allow operations on character strings represented as vectors of characters and allow efficient Boolean vector operations.

Transfers between arrays or records of identical logical structure are necessary to permit efficient run time conversion from one object representation to another, as might be done when data is packed to reduce peripheral storage requirements and I/O transfer times but need to be unpacked locally to minimize processing costs.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversions operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

Implicit type conversions which represent changes in the value of data items without an explicit indicator in the program, are not only error prone but can result in unexpected run time overhead.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

Because ranges do not form closed systems, range validation is not possible at compile time (e.g., "I:=I+1" may be a range error). At best, the compiler might point out likely range errors. (This requirement is optional for hardware installations which do not have overflow detection).

B10. The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

Whether the referenced "files" are real or virtual and whether they are hardware devices, I/O channels or logical files depends on the object machine configuration and on the details of its operating system if present. But in any programming system I/O operations ultimately reduce to sending or receiving data and/or control information to a file or to a device controller. These can be made accessible in a HOL in an abstract form through a small set of generic I/O operations (like "read" and "write," with appropriate device and exception parameters). Note that devices and files are similar in many respects to types, so additional language features may not be required to satisfy this requirement. This requirement, in conjunction with requirement E1, permits user definition of unique equipment and its associated I/O operations as data types within the syntactic and semantic framework provided by the generic operations.

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

As with any data type, power sets will be useful only if there are operations which can create, select and interrogate them. Note that this provides only a very

special class of sets but one which is very useful for computations on sets of indicators, flags, and similar devices in monitoring and control applications. More general sets if desired, must be defined using the type definition facilities.

C. EXPRESSIONS AND PARAMETERS

1. Side Effects
2. Operand Structure
3. Expressions Permitted
4. Constant Expressions
5. Consistent Parameter Rules
6. Type Agreement in Parameters
7. Formal Parameter Kinds
8. Formal Parameter Specifications
9. Variable Numbers of Parameters

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

This is a semantic restriction on the evaluation order of arguments to expressions. It provides an explicit rule (i.e., left-to-right) for order of argument evaluation, but allows the implementations to alter the actual order in any way which does not change the effect. This provides the user with a simple rule for determining the effects of interactions among argument evaluations without imposing a strict rule on compilers which are sophisticated enough to detect potential side-effects and optimize through reordering of arguments when the evaluation order does not affect the result. Control operations (e.g., conditional and iterative control structures), of course, must be exceptions to this general rule since control operations are in fact those operations which specify the sequencing and evaluation rules for their arguments.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

Care must be taken to ensure that the operator/operand structure of expressions is not psychologically ambiguous (i.e., to guarantee that the parse implemented by the language is the same as intended by the programmer and understood by those reading the program). This kind of problem can be minimized by having few precedence levels and parsing rules by allowing explicit parentheses to specify the intended execution order, and by requiring explicit parentheses when the execution order is of significance to the result within the same precedence level (e.g., "X divided by Y divided by Z" and "X divided by Y multiplied by Z"). The user will not be able to define new operator precedence rules nor change the precedence of existing operators.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

This is an example of not imposing arbitrary restrictions and special case rules on the user of the source language. Special mention is made here only because so many languages do restrict the form of expressions. FORTRAN, for example, has a list of seven different syntactic forms for subscript expressions, instead of allowing all forms of arithmetic expressions.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

The ability to write constant expressions in programs has proven valuable in languages with this capability, particularly with regard to program readability and in avoiding programmer error in externally evaluating and transcribing constant expressions. They are most often used in declarations. There is no need, however, that constant expressions impose run time costs for their evaluation. They can be evaluated once at compile time or if this is inconvenient because of incompatibilities between the host and object machines, the compiler can generate code for their evaluation at load time. In any case, the resulting value should be the same (at least within the stated precision) regardless of the object machine (see D2). Allowing constant expressions in place of constants can improve the clarity, correctness and maintainability of programs and does not impose any run time costs.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handing, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contributes to ease of learning,

implementing and using a language; allows the user to concentrate on the programming task instead of the language; and leads to more readable, understandable, and predictable programs.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

Type transfers hidden in procedure calls with incompatible formal and actual parameters whether intentional or accidental have long been a source of program errors and of programs which are difficult to maintain. On the other hand, there is no reason why the subscript ranges for arrays cannot be passed as part of the arguments. Some notations permit such parameters to be implicit on the call side. Formal parameters of a union type will be considered conformable to actual parameters of any of the component types.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

The first class of data parameter acts as a constant within the procedure body and cannot be assigned to nor changed during the procedures execution; its corresponding actual parameter may be any legal expression of the desired type and will be evaluated once at the time of call. The second class of data parameter renames the actual parameter which must be a variable, the address of the actual parameter variable will be determined by (or at) the time of call and unalterable during execution of the procedure, and assignment (or reference) to the formal parameter name will assign (or access) the variable which is the actual parameter. These are the only two widely used parameter passing mechanisms for data and the many alternatives (at least 10 have been suggested) add complexity and cost to a language without sufficiently increasing the clarity or power. A language with exception handling capability must have a way to pass control and related data through procedure call interfaces. Exception handling control parameters will be specified on the call side only when needed. Actual procedure parameters will be restricted to those of similar (explicit or implicit) specification parts.

C8. Specification of the type, range, precision, dimension, scale and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

Optional formal parameter specification permits the writing of generic procedures which are instantiated at compile time by the characteristics of their actual parameters. It eliminates the need for compile time "type" parameters. This generic procedure capability, for example, allows the definition of stacks and queues and their associated operations on data of any given type without knowing the data type when the operations are defined.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as "print," generalizations of operations which are both commutative and associative such as "max" and "min," and repetitive application of the same binary operation such as the Lisp "list" operation. The use of variable number of argument operations need not and will not cause relaxation of any compile time checks, require use of multiple entry procedures allow the number of actual parameters to vary at run time, nor require special calling mechanisms. If the parameters which can vary are limited to a program specified type treated as any other argument on the call side and as elements of an array within the procedure definition, full type checking can be done at compile time. There will be not prohibition on writing a special case of a procedure for a particular number of arguments.

D. VARIABLES, LITERALS AND CONSTANTS
1. Constant Value Identifiers
2. Numeric Literals
3. Initial Values of Variables
4. Numeric Range and Step Size
5. Variable Types
6. Pointer Variables

D1. The user will have the ability to associate constant values of any type with identifiers.

The use of identifiers to represent constant values has often made programs more readable, more easily modifiable and less prone to error when the value of a constant is changed. Associating constant values with an identifier is preferable to assigning the value to a variable because it is then clearly marked in the program as a constant, can be automatically checked for unintentional changes, and often can have a more efficient object representation.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

Literals are needed for all atomic data types and should be provided as part of the language definition for built-in types. Regardless of the source of the data and regardless of the object machine the value of constants should be the same. For integers it should be exact and for reals it should be the same within the specified precision. Compiler writers, however, would disagree. They object to this requirement on two grounds: that it is too costly if the host and object machines are different and that it is unnecessary if they are the same. In fact, all costs are at compile time and must be insignificant compared to the life time costs resulting from object cope containing the wrong constant values. As for being unnecessary, there have been all too many cases of different values from program and data literals on the same machine because the compile time and run time conversion packages were different and imprecise.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

The ability to initialize variables at the time of their allocation will contribute to program clarity, but a requirement to do so would be an arbitrary and sometimes costly decision to the user. Default initial values on the other hand, contribute to neither program clarity nor correctness and can be even more costly at run time. It is usually a programming error if a variable is accessed before it is initialized. It is desirable that the translator give a warning when a path between the declaration and use of a variable omits initialization. Whether a variable will be assigned is in general an unsolvable problem, but it is sometimes determinable whether assignments occur on potential paths. In the case of arrays, it it possible at compile time only to determine that some components (but not necessarily which) have been initialized. There will be provision (at user option) for run time testing for initialization.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

Range specifications are a special form of assertion. They aid in understanding and determining the correctness of programs. They can also be used as additional information by the compiler in deciding what storage and allocation to use (e.g., half words might be more efficient for integers in the range 0 to 1000). Range specifications also offer the opportunity for the translator to insert range tests automatically for run time or debug time validation of the program logic. With the ranges of variables specified in the program, it becomes possible to perform many subscript bounds checks at compile time. These bounds, checks, however, can be only as valid as the range specifications which cannot in general be validated at compile time. Range specifications on approximate valued variables (usually with floating point implemetation) also offer the possibility of their implementation using fixed point hardware.

D5. The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

There should not be any arbitrary restrictions on the structure of data. This permits arrays to be components of records or arrays and permits records to be components of arrays.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

Assignment to a pointer variable will mean that the variable's name is to act as an additional label (or reference) on the datum being assigned. Assignment to a nonpointer variable will mean that the variable's name it to label a copy of the object being assigned. For data without alterable component structure or alterable component values, there is no functional difference between reference to multiple copies and multiple references to a single copy. Consequently, pointer/nonpointer will be a property only of variables for composite types and of composite array and record components Because the pointer/nonpointer property applies to all variables of a given type, it will be specified as part of the type definition. The use of pointers will be kept safe by prohibiting pointers to data structures whose allocation scope is narrower than that of the pointer variable.

Such a restriction is easily enforced at compile time using hierarchical scope rules providing there is no way to dynamically create new instances of the data type. In the latter case, the dynamically created data can be allocated with full safety using a (user or library defined) space pool which is either local (i.e., own) or global to the type definition. If variables of a type do not have the pointer property then dynamic storage allocation would be required for assignment unless their size is constant and known at the time of variable allocation. Thus, the nonpointer property will be permitted only for types (a) whose data have a structure and size which is constant in the type definition or (b) which manage the storage for their data as part of the type definition. Because pointers are often less expensive at run time than nonpointers and are subject to fewer restrictions, the specification of the nonpointer property will be explicit in programs (this is similar to the Algol-60 issue concerning the explicit specification of "value" (i.e., nonpointer) and "name" (i.e. pointer). The need for pointers is obvious in building data structures with shared or recursive substructures; such as, directed graphs, stacks, queues, and list structures. Providing pointers as absolute address types, however, produces gaps in the type checking and scope mechanisms. Type and access restricted pointers will provide the power of general pointers without their undesirable characteristics.

E. DEFINITION FACILITIES
1. User Definitions Possible
2. Consistent Use of Types
3. No Default Declarations
4. Can Extend Existing Operators
5. Type Definitions
6. Data Defining Mechanisms
7. No Free Union or Subset Types
8. Type Initialization

E1. The user of the language will be able to define new data types and operations within programs.

The number of specialized capabilities needed for a common language is large and diverse. In many cases, there is no consensus as to the form these capabilities should take in a programming language. The operational requirements dictating specific specialized language capabilities are volatile and future needs cannot always be foreseen. No language can make available all the features useful to the broad spectrum of military applications, anticipate future applications and requirements or even provide a universally "best" capability in support of a single application area. A common language needs capability for growth. It should contain all the power necessary to satisfy all the applications and the ability to specialize that power to the particular application task. A language with defining facilities for data and operations often makes it possible to add new application-oriented structures and to use new programming techniques and mechanisms through descriptions written entirely within the language. Definitions will have the appearance and costs of features which are built into the language while actually being catalogued accessible application packages. The operation definition facility will include the ability to define new infix operators (but see H2 for restrictions). No programming language can be all things to all people, but a language with data and operation definition facilities can be adapted to meet changing requirements in a variety of areas.

The ability to define data and operations is well within the state of the art. Operation definition facilities in the form of subroutines have been available in all general purpose programming languages since at least the time of early FORTRANs. Data definition facilities have been available in a variety of programming languages for almost 10 years and reached their peak with a large number of extensible languages(Stephen A. Schuman (Ed.) Proceedings of the International Symposium on Extensible Languages, SIGPLAN Notices, Vol. 6, No. 12, December 1971. Also, C. Christensen and C.J. Shaw (Ed.), Proceedings of the Extensible Language Symposium, SIGPLAN Notices 4, August 1969.) (over 30) in 1968 and shortly thereafter. A trend toward more abstract and less machine-oriented data

specification mechanisms has appeared more recently in PASCAL(Niklaus Wirth, "An Assessment of the Programming Language PASCAL, "Proceedings of the International Conference on Reliable Software 21-23 April 1973, p. 23-30). Data type definitions, with operations and data defined together, are used in several languages including SIMULA-67(Jacob Palme, "SIMULA as a Tool for Extensible Program Products, "SIGPLAN Notices, Vol. 9, No. 4, February 1974). On the other hand, there is currently much ferment as to what is the proper function and form of data type definitions.

E2. The "use" of defined types will be indistinguishable from built-in types.

Whether a type is built-in or defined within the base will not be determinable from its syntactic and semantic properties. There will be no ad hoc special cases nor inconsistent rules to interfere with and complicate learning, using and implementing the language. If built-in features and user defined data structures and operations are treated in the same way throughout the language so that the base language, standard application libraries and application programs are treated in a uniform manner by the user and by the translator, then these distinctions will grow dim to everyone's advantage. When the language contains all the essential power, when few can tell the difference between the base language and library definitions, and when the introduction of new data types and routines does not impact the compiler and the language standards, then there is little incentive to proliferate languages. Similarly, if typed definitions are processed entirely at compile time and the language allows full program specification of the internal representation, there need be no penalty in run time efficiency for using defined types.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

As programmers, we should not expect the translator to write our programs for us (at least in the immediate future). If we somehow know that the translator's default convention is compatible with our needs for the case at hand we should still document the choice so others can understand and maintain our programs. Neither should we be able to delay definitions (possibly forget them) until they cause trouble in the operational system. This is a special case of requirement I1.

E4. The user will be able, within the source language, to extend existing operators to new data types.

When an operation is an abstraction of an existing operation for a new type or is a generalization of an existing operation, it is inconvenient, confusing and misleading to use any but the existing operator symbol or function named. The translator will not assume that commutativity of bulit-in operations is preserved by extensions, and any assumptions about the associativity of built-in or extended operations will be ignored by the translator when explicit parentheses are provided in an expression.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

Types define abstract data objects with special properties. The data objects are given a representation in terms of existing data structures, but they are of little value until operations are available to take advantage of their special properties. When one obtains access to a type, he needs its operations as well as its data. Numeric data is needed in many applications but is of little value to any without arithmetic operations. The definable operations will include constructors, selectors, predicates, and type conversions.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

The above list comprises a currently known set of useful definitional mechanisms for data types which do not require run time support, as do garbage collection and dynamic storage allocation. In conjunction with pointers (see D6), they provide many of the mechanisms necessary to define recursive data structures and efficient sparse data structures.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

Free union adds no new power not provided by discriminated union, but does require giving up the security of types in return for programmer freedom. Range and

subset specifications on variables are useful documentation and debugging aids, but will not be construed as types. Subsets do not introduce new properties or operations not available to the superset and often do not form a closed system under the superset operations. Unlike types, membership in subsets can be determined only at run time.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

It is often necessary to do bookkeeping or to take other special action when variables of a given type are allocated or deallocated. The language will not limit the class of definable types by withholding the ability to define those actions. Initialization might take place once when the type is allocated (i.e., in its allocation scope) and would be used to set up the procedures and initialize the variables which are local to the type definition. These operations will be definable in the encapsulation housing the rest of the type definition.

F. SCOPES AND LIBRARIES
1. Separate Allocation and Access Allowed
2. Limiting Access Scope
3. Compile Time Scope Determination
4. Libraries Available
5. Library Contents
6. Libraries and Compools Indistinguishable
7. Standard Library Definitions

F1. The language will allow the user to distinguish between scope of allocation and scope of access.

The scope of allocation or lifetime of a program structure is that region of the program for which the object representation of the structure should be present. The allocation scope defines the program scope for which own variables of the structure must be maintained and identifies the time for initialization of the structure. The access scope defines the regions of the program in which the allocated structure is accessible to the program and will never be wider than the allocation scope. In some cases the user may desire that each use of a defined program structure be independent (i.e., the allocation and accessing scopes would be identical). In other cases, the various accessing scopes might share a common allocation of the structure.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

Limited access specified in a type definition is necessary to guarantee that changes to data representations and to management routines which purportedly do not affect the calling programs are in fact safe. By rigorously controlling the set of operations applicable to a defined type, the type definition guarantees that no external use of the type can accidentally or intentionally use hidden nonessential properties of the type. Renaming separately defined programming components is necessary to avoid naming conflicts when they are used.

Limited access on the call side provides a high degree of safety and eliminates nonessential naming conflicts without limiting the degree of accessibility which can be built into programs. The alternative notion, that all declarations which are external to a program segment should have the same scope, is inconvenient and

costly in creating large systems which are composed from many subsystems because it forces global access scopes and the attendant naming conflicts on subsystems not using the defined items.

F3. The scope of identifiers will be wholly determined at compile time.

Identifiers will be declared at the beginning of their scope and multiple use of variable names will not be allowed in the same scope. Except as otherwise explicitly specified in programs, access scopes will be lexically embedded with the most local definition applying when the same identifier appears at several levels. The language will use the above lexically embedded scope rules for declarations and other definitions of identifiers to make them easy to recognize and to avoid errors and ambiguities from multiple use of identifiers in a single scope.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

A simple base alone is not sufficient for a common language. Even though in theory such a language provides the necessary power and the capability for specialization to particular applications, the users of the language cannot be expected to develop and support common libraries under individual projects There will be broad support for libraries common to users of well recognized application areas. Application libraries will be developed as early as possible.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

The usefulness of a language derives primarily from the existence and accessibility of specialized application-oriented data and operations. Whether a library should contain source or object code is a question of implementation efficiency and should not be specified in the definition of the source language, but the source language description will always be available. It should be remembered, however, that interfaces cannot be validated at program assembly time without some equivalent of their source language interface specifications, that object modules are machine-dependent and, therefore, not portable, that source code is often more compact than object code, and that compilers for simple languages can sometimes

compile faster than a loader can load from relocatable object programs. Library routines written on other languages will not be prohibited provided the foreign routine has object code compatible with the calling mechanisms used in the Common HOL and providing sufficient header information (e.g., parameter types, form and number) is given with the routine in Common HOL form to permit the required compile time checks at the interface.

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

Compools have proven very useful in organizing and controlling shared data structures and shared routines. A similar mechanism will be available to manage and control access to related library definitions.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

The convenience, ease of use and savings in production and maintenance costs resulting from using high order languages come from being able to use specialized capabilities without building them from scratch. Thus, it is essential that high level capabilities be supplied with the language. The idea is not to provide all the many special cases in the language, but to provide a few general case . hich will cover the special cases.

There is currently little agreement on standard operating system, I/O, or file system interfaces. This does not preclude support of one or more forms for the near term. For the present the important thing is that one be chosen and made available as a standard supported library definition which the user can use with confidence.

G. CONTROL STRUCTURES
1. Kinds of Control Structures
2. The Go To
3. Conditional Control
4. Iterative Control
5. Routines
6. Parallel Processing
7. Exception Handling
8. Synchronization and Real Time

G1. *The language will provide structured control mechanisms for* sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

These mechanisms, hopefully, provide a spanning set of control structures. The most appropriate operations in several of these areas is an open question. For the present, the choice will be a spanning set of composable control primitives each of which is easily mapped onto object machines and which does not impose run time charges when it is not used. Whether parallel processing is real (i.e., by multiprocessing) or is synthesized on a single sequential processor, is determined by the object machine, but if programs are written as if there is true parallel processing (and no assumption about the relative speeds of the processors) then the same results will be obtained independent of the object environment.

It is desirable that the number of primitive control structures in the language be minimized, not by reducing the power of the language, but by selecting a small set of composable primitives which can be used to easily build other desired control mechanisms within programs. This means that the capabilities of control mechanisms must be separable so that the user need not pay either program clarity or implementation costs for undesired specialized capabilities. By these criteria, the Algol-60 "FOR" would be undesirable because it imposes the use of a loop control variable, requires that there be a single terminal condition and that the condition be tested before each iteration. Consequently, "FOR" cannot be composed to build other useful iterative control structures (e.g., FORTRAN "DO"). The ability to compose control structures does not imply an ability to define new control operations and such an ability to define new control operations, and such an ability is in conflict with the limited parameter passing mechanisms of C7.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

The "GO TO" is a machine level capability which is still needed to fill in any gaps which might remain in the choice of structured control primitives, to provide compatibility for translitterating programs written in older languages, and because of the wide familiarity of current practitioners with its use. The language should not, however, impose unnecessary costs for its presence. The "GO TO" will be limited to explicitly specified program labels at the same scope level. Neither should the language provide specialized facilities which encourage its use in dangerous and confusing ways. Switches, designational expressions, label variables, label parameters and numeric labels are not desired. Switches here refer to the unrestricted switches which are generalizations of the "GO TO" and not to case statements which are a general form for conditionals(see G3). This requirements should not be interpreted to conflict with the specialized form of control transfer provided by the exception handling control structure of G7.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

The conditional control operations will be fully partitioned (e.g., an "ELSE" clause must follow each "IF THEN") so the choice is clear and explicit in each case. There will be some general form of conditional which allows an arbitrary computation to determine the selected situation (e.g., Zahn's device(Donald E. Knuth, "Structured Programming with go to Statements," ACM Computer Surveys, Vol. 6, No. 4, December 1974) provides a good solution to the general problem). Special mechanisms are also needed for the more common cases of the Boolean expression (e.g., "IF THEN ELSE") and for value or type discrimination (e.g., "CASE" on one of a set of values or subtype of a union).

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables t be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

In its most general form, a programmed loop is executed repetitively until some computed predicate becomes true. There may be more than one terminating predicate, and they might appear anywhere in the loop. Specialized control structures (e.g., "WHILE DO") have been used for the common situation in which the

termination conditions precedes each iteration. The most common case is termination after a fixed number of iterations and a specialized control structure should be provided for that purpose (e.g., FORTRAN "DO" or Algol-60 "FOR"). A problem which arises in many programming languages is that loop control variables are global to the iterative control and thus, will have a value after loop termination, but that value is usually an accident of the implementation. Specifying the meaning of control variables after loop termination in the language definition resolves the ambiguity but must be an arbitrary decision which will not aide program clarity or correctness, and may interfere with the generation of efficient object code. Loop control variables are by definition variables used to control the repetitive execution of a programmed loop and as such will be local to the loop body, but at loop termination it will be possible to pass their value (or any other computed value) out of the loop, conveniently and efficiently.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

Recursion is desirable in many applications because it contributes directly to their elegance and clairty and simplifies proof procedures. Indirectly, it contributes to the reliability and maintainability of some programs. Recursion is required in order to avoid unnecessarily opaque, complex and confusing programs when operating on recursive data structures. Recursion has not been widely used in DoD software because many programming languages do not provide recursion, practitioners are not familiar with its use, and users fear that its run time costs are to high. Of these, only the run time costs would justify its exclusion from the language.

A major run time cost often attributed to recursion is the need for the presence of a set of "display" registers which are used to keep track of the addresses of the various levels of lexically imbedded environments and which must be managed and updated at run time. The display, however, is necessary only in programs in which routines access variables which are global to their own definition, but local to a more global recursive procedure. This possibility can easily be removed by prohibiting the definition of procedures within the body of a recursive procedure. The utility of such a combination of capabilities is very questionable, and this single restriction will eliminate all added execution costs for nonrecursive procedures in programs which contain recursive procedures.

As with any other facility of the language, routines should be implemented in the most efficient manner consistent with their use and the language should be designed so that efficient implementations are possible. In particular, the most possible regardless of whether the language or even the program contains recursive

procedures. When any routine makes a procedure call as its last operation before exit (and this is quite common for recursive routines) the implementation might use the same data area for both routines, and do a jump to the head of the called procedure thereby saving much of the overhead of a procedure call and eliminating a return. The choice between recursive and nonrecursive routines involves trade-offs. Recursive routines can aid program clarity when operating on recursive data, but can detract from clarity when operating on iterative data. They can increase execution time when procedure call overhead is greater than loop overhead and can decrease execution times when loop overhead is the more expensive. Finally, program storage for recursive routines is often only a small fraction of that for a corresponding iterative procedure, but the data storage requirements are often much larger because of the simultaneous presence of several activations of the same procedure.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

A parallel processing capability is essential in embedded computer applications. Programs must send data to, receive data from, and control many devices which are operating in parallel. Multiprogramming (a form of pseudo paralell processing) is necessary so that many programs within a system can meet their differing real time constraints. The parallel processing capability will minimally provide the ability to define and call parallel processing and the ability to gain exclusive use of system resources in the form of data structures, devices and pseudo devices. This latter ability satisfies one of the two needs for synchronization of parallel processes. The other is required in conjunction with real time constraints (see G8).

The parallel processing capability will be defined as true parallel (as opposed to coroutine) primitives, but with the understanding that in most implementations the object computer will have fewer processors (usually one) than the number of parallel paths specified in a program. Interleaved execution in the implementation may be required.

The parallel processing features of the language should be selected to eliminate any unnecessary overhead associated with their use. The costs of parallel processes are primarily in run time storage management. As with recursive routines most accessing and storage management problems can be eliminated by prohibiting complex interactions with other language facilities where the combination has little if any utility. In particular, it will not be possible to define a parallel routine within the

body of a recursive routine and it will not be possible to define any routine including parallel routines within the body of those parallel routines which can have multiple simultaneous activations. If the language permits several simultaneous activations of a given parallel process then it might require the user to give a upper bound on the number which can exist simultaneously. The latter requirement is reasonable for parallel processes because it is information known by the programmer and necessary to the maintainer, because parallel processes cannot normally be stacked, and because it is necessary for the compilation of efficient programs.

G7. The exception handing control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

It is essential in many aplications that there be no program halts beyond the user's control. The user must be able to specify the action to be taken on any exception situation which might occur within his program. The exception handling mechanism will be parameterized so data can be passed to the recovery point. Exception situations might include arithmetic overflow, exhaustion of available space, hardware errors, any user defined exceptions and any run time detected programming error.

The user will be able to write programs which can get out of an arbitrary nest of control and intercept it at any embedding level desired. The exception handling mechanism will permit the user to specify the action to be taken upon the occurrence of a designated exception within any given access scope of the program. The transfers of control will, at the users option, be either forward in the program (but never to a narrower scope of access or out of a procedure) or out of the current procedure through its dynamic (i.e., calling structure. The latter form requires an exception handling formal parameter class (see C7).

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

When parallel or pseudo parallel paths appear in a program it must be possible to specify their relative priorities and to synchronize their executions. Synchronization can be done either through exclusive access to data (see G6) or through delays terminated by designated situations occurring within the program.

These situations should include the elapse of program specified time intervals, occurrence of hardware interrupts and those designated in the program. There will be no implicit evaluation of program determined situations. Time delays will be program specifiable for both real and simulated times.

H. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics
2. No Syntax Extensions
3. Source Character Set
4. Identifiers and Literals
5. Lexical Units and Lines
6. Key Words
7. Comment Conventions
8. Unmatched Parentheses
9. Uniform Referent Notation
10. Consistency of Meaning

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

Clarity and readability of programs will be the primary criteria for selecting a syntax. Each of the above points can contribute to program clarity. The use of free format, mnemonic identifiers and conventional forms allows the programmer to use notations which have their familiar meanings to put down his ideas and intentions in the order and form that humans think about them, and to transfer skills he already has to the solution of the problem at hand. A simple uniform language reduces the number of cases which must be dealt with by anyone using the language. If programs are difficult for the translator to parse they will be difficult for people. Similar things should use the same notations with the special case processing reserved for the translator and object machine. The purpose of mnemonic identifiers and key words is to be informative and increase the distance between lexical units of programs. This does not prevent the use of short identifiers and short key words.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.

If the user can change the syntax of the language, then he can change the basic character and understanding of the language. The distinction between

semantic extensions and syntactic extensions is similar to that between being able to coin new words in English or being able to move to another natural language. Coining words requires learning those new meanings before they can be used, but at the same time increases the power the language for some application areas. Changing the grammar, (e.g., Franglais, the use of French grammar with interspersed English words) however, undermines the basic understanding of the language itself, changes the mode of expression, and removes the commonalities which obtain between various specializations of the language. Growth of a language through definition of new data and operations and the introduction of new words and symbols to identify them is desirable, but there should be no provision for changing the grammatical rules of the language. This requirement does not conflict with E4 and does not preclude associating new meanings with existing infix operators.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

A common language should use notations and a character set convenient for communicating algorithms, programs, and programming techniques among its users. On the other hand, the language should not require special equipment (e.g., card readers and printers) for its use. The use of the 64 character ASCII subset will make the language compatible with the federal information processing standard 64 character set, FIPS-1, which has been adopted by the U.S.A. Standard code for Information Interchange (USASCII). The language definition will specify the translation from the publication language into the restricted character set.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

Lexical units of the language should be defined in a simple uniform and easily understood manner. Some possible break characters are the space(W. Dijkstra, coding examples in Chapter I, "Notes in Structured Programming," in Structured Programming by O.-J. Dahl, E. W. Dijkstra and C.A.R. Moore, Academic Press, 1972. & Thomas A. Standish, "A Structured Program to Play Tic-Tac-Toe," notes for Information and Computer Science 3 course at Univ. of California-Irvine, October 1974) (i.e., any number of spaces or end-of-line), the underline and the tilde. The space cannot be used if identifiers and user defined infix operators are lexically indistinguishable, but in such a case the formal grammar for the language would be ambiguous (see H1). A literal break character contributes to the readability of

programs and makes the entry of long literals less error prone. With a space as a break character one can enter multipart (i.e., more than one lexical unit) identifiers such as "REAL TIME CLOCK" or long literals, such as, "3.14159 26535 89793." Use of a break can also be used to guarantee that missing quote brackets on character literals do not cause errors which propagate beyond the net end-of-line. the language should require separate quoting of each line of a long literal: "This is a long" "literal string".

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

Many elementary input errors arise at the end of lines. Programs are input on line oriented media but the concept of end-of-line is foreign to free format text. Most of the error prone aspects of end-of-line can be eliminated by not allowing lexical units to continue over lines. The sometimes undesirable effects of this restriction can be avoided by permitting identifiers and literals to be composed from more than one lexical unit (see H4) and by evaluating constant expressions at compile time (see C4).

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

By key words of the language are meant those symbols and strings which have special meaning in the syntax of programs. They introduce special syntactic forms such as are used for control structures and declarations or the are used as infix operators, or as some form of parenthesis. Key words will be reserved, that is unusable as identifiers, to avoid confusion and ambiguity. Key words will be few in number because each new key word introduces another case in the parsing rules and thereby adds to complexity in understanding the language, and because large numbers of key words inconvenience and complicate the programmer's task of choosing informative identifiers. Key words should be concise, but being information is more important than being short. A major exception is the key word introducing a comment; it is the comment and not its key word which should do the informing. Finally, there will be no place in a source language program in which a key word can be used in place of an identifier. That is, functional form operations and special data items built into the language or accessible as a standard extension will not be treated as key words but will be treated as any other identifier.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

These are all obvious points which will encourage the use of comments in programs and avoid their error prone features in some existing languages. Comments anywhere reasonable in a program will not be taken to mean that they can appear internal to a lexical unit, such as, an identifier, key word, or between the opening and closing brackets of a character string. One comment convention which nearly meets these criteria is to have a special quote character which begins comments and with either the quote or an end-of-line ending each comment. This allows both embedded and line-oriented comments.

H8. The language will not permit unmatched parentheses of any kind.

Some programming languages permit closing parentheses to be omitted. If, for example, a program contained more "BEGINs" than "ENDs" the translator might insert enough "ENDs" at the end of the program to make up the difference. This makes programs easier to write because it sometimes saves writing several "ENDs" at the end of programs and because it eliminates all syntax errors for missing "ENDs." Failure to require proper parentheses matching makes it more difficult to write correct programs. Good programming practice requires that matching parentheses be included in programs whether or not they are required by the language. Unfortunately, if they are not required by the language then there can be no syntax check to discover where errors were made. The language will require full parentheses matching. This does not preclude syntactic features such as "case x of s1, s2...sn end case" in which "end" is paired with a key word other than "begin." Nor does it alone prohibit open forms such as "if-then-else-."

H9. There will be a uniform referent notation.

The distinction between function calls and data reference is one of representation, not of use. Thus, there will be no language imposed syntactic distinction between function calls and data selection. If, for example, a computed function is replaced by a lookup table there should be no need to change the calling program. This does not preclude the inclusion of more than one referent notation.

H10. No language defined symbols appearing in the same context will have essentially different meanings.

This contributes to the clarity and uniformity of programs, protects against psychological ambiguity and avoids some error prone features of extant languages. In particular, this would exclude the use of = to imply both assignment and equality, would exclude conventions implying that parenthesized parameters have special semantics (as with PL/1 subroutines), and would exclude the use of an assignment operator for other than assignment (e.g., left hand side function calls). It would not, however, require different operator symbols for integer, real or even matrix arithmetic since these are in fact special cases of the same abstract operations and would allow the use of generic functions applicable to several data types.

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic
2. Object Representation Specifications Optional
3. Compile Time Variables
4. Conditional Compilation
5. Simple Base Language
6. Translator Restrictions
7. Object Machine Restrictions

11. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

The only alternative is implementation dependent defaults with the translator determining the meaning of programs. What a program does, should be determinable from the program and the defining documentation for the programming language. This does not require that binding of all program properties be local to each use. Quite the contrary, it would, for example, allow automatic definition of assignment for all variables or global specification of precision. What it does require is that each decision be explicit: in the language definition, global to some scope, or local to each use. Omission of any selection which affects the program logic will be treated as an error by the translator.

12. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

The language should be oriented to provide a high degree of management control and visibility to programs and toward self-documenting programs with the programmer required to make his decisions explicit. On the other hand, the programmer should not be forced to overspecify his programs and thereby cloud their logic, unnecessarily eliminate opportunities for optimization, and misrepresent arbitrary choices as essential to the program logic. Defaults will be allowed, in fact, encouraged in don't care situations. Such defaults will include data representations (see J4), open vs. closed subroutine calls (see J5), and reentrant vs. nonreentrant code generation.

13. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

When a language has different host and object machines and when its compilers can produce code for several configurations of a given machine, the programmer should be able to specify the intended object machine configuration. The user should have control over the compile time variables used in his program. Typically they would be associated with the object computer model, the memory size, special hardware options, the operating system if present, peripheral equipment or other aspects of the object machine configuration. Compile time variables will be set outside the program, but available for interrogation within the program (see I4 and C4).

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

An environmental inquiry capability permits the writing of common programs and procedures which are specialized at compile time by the translator as a function of the intended object machine configuration or of other compile time variables (see I3). This requirement is a special case of evaluation of constant expressions at compile time (see C4). It provides a general purpose capability for conditional compilation.

15. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capablity not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

The capabilities available in any language can be partitioned into two groups, those which are definable within the base and those which provide an essential primitive capability of the language. The smaller and simpler the base the easier the language will be to learn and use. A clearly delineated base with features not in the base defined in terms of the base, will improve the ease and efficiency of learning, implementing and maintaining the language. Only the base need be implemented to make the full source language capability available.

Base features will provide relatively low level general purpose capabilities not yet specialized for particular applications. There will be no prohibition on a translator incorporating speciaized optimizations for particular extensions. Any extension provided by a translator will, however, be definable within the base language using the built-in definition facilities. Thus, programs using the extension will be translatable by any compiler for the language but not necessarily with the same object efficiency.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels in expressions, or the number of identifiers in programs are determined by the translator and not by the object machine. Ideally, the limits should be set so high that no program (save the most abrasive) encounters the limits. In each case, however: (a) some limit must be set, (b) whatever the limit, it will impose on some and therefore must be known by the users of the translator, (c) letting each translator set its own limits means that programs will not be portable, (d) setting the limits very high requires that the translator be hosted only on large machines and (e) quite low limits do not impose significantly on either the power of the language or the readability of programs. Thus, the limits should be set as part of the language definition. They should be small enough that they do not dominate the compiler and large enough that they do not interfere with the usefulness of the language. If they were set at say the 99 percent level based on statistics from existing DoD computer programs the limits might be a few hundred for numbers of identifiers and less than ten in the other cases mentioned above.

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

Limits on the amount of run time storage, access to specialized peripheral equipments, use of special hardware capabilities and access to real time clocks are dependent on the object machine and configuration. The translator will report when a program exceeds the resources or capabilities of the intended object machine but will not build in arbitrary limits of its own.

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES
1. Efficient Object Code
2. Optimizations Do Not Change Program Effect
3. Machine Language Insertions
4. Object Representation Specifications
5. Open and Closed Routine Calls

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

The base language and library definitions might contain features and capabilities which are not needed by everyone, or at least, not be everyone all the time. The language should not force programs to require greater generality than they need. When a program does not use a feature or capability it should pay no run time cost for the feature being in the language or library. When the full generality of a feature is not used, only the necessary (reduced) cost should be paid. Where possible, language features (such as, automatic and dynamic array allocation, process scheduling, file management and I/O buffering) which require run time support packages should be provided as standard library definitions and not as part of the base language . The user will not have to pay time and space for support packages he does not use. Neither will there be automatic movement of programs or data between main store and backing shore which is not under program control (unless the object machine has virtual memory with underlying management beyond the control of all its users). Language features will result in special efficient object codes when their full generality is not used. A large number of special cases should compile efficiently. For example, a program doing numerical calculations on unsubscripted real variables should produce code no worse than FORTRAN. Parameter passing for single argument routines might be implemented much less expensively than multiple argument routines.

One way to reduce costs for unneeded capabilities is to have a base language whose data structures and operations provide a single capability which is composable and has a straight-forward implementation in the object code of conventional architecture machines. If the base language components are easily composable they can be used to construct the specialized structures needed by specific applications, if they are simple and provide a single capability they will not force the use of unneeded capabilities in order to obtain needed capabilities, and if they are compatible with the features normally found in sequential uniprocessor digital computers with random access memory they will have near minimum or at least low cost implementation on many object machines.

J2. Any optimizations performed by the translator will not change the effect of the program.

More simply, the translator cannot give up program reliability and correctness, regardless of the excuse. Note that for most programming languages there are few known safe optimizations and many unsafe ones. The number of applicable safe optimizations can be increased by making more information available to the compiler and by choosing language constructs which allow safe optimizations. This requirement allows optimization by code motion providing that motion does not change the effect of the program.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

It is difficult to be enthusiastic about machine language insertions. They defeat the purpose of machine independence constrain the implementation techniques complicate the diagnostics, impair the safety of type checking, and detract from the reliability, readability, and modifiability of programs. The use of machine language insertions is particularly dangerous in multiprogramming applications because they impair the ability to exclude, "a priori," a large class of time-dependent bugs. Rigid enforcement of scope rules by the compiler in real time applications is a powerful tool to ensure that one sequential process will not interfere with others in an uncontrolled fashion. Similarly, when several independent programs are executed in an interleaved fashion, the correct execution of each may depend on the others not having improperly used machine language insertions.

Unfortunately machine language insertions are necessary for interfacing special purpose devices, for accessing special purpose hardware capabilities, and for certain code optimizations on time critical paths. Here we have an example of Dijkstra's dilemma in which the mismatch between high level language programming and the underlying hardware is unacceptable and there is no feasible way to reject the hardware. The only remaining alternative is to "continue bit pushing in the old way, with all the known ill effects." Those ill effects can, however, be constrained to the smallest possible perimeter in practice if not in theory. The ability to enter machine language should not be used as an excuse to exclude otherwise needed facilities from the HOL; the abstract description of programs in the HOL should not require the use of machine language insertions. The semantics of machine language insertions will be determinable from the HOL definition and the object machine description alone and not dependent on the translator characteristics Machine language insertions will be encapsulated so they can be easily recognized and so that it is clear which variables and program identifiers are accessed within the insertion. The machine language insertions will be permitted only within the body of compile

time conditional statements (see I4) which depend on the object machine configuration (see I3). They will not be allowed interspersed with executable statements of the source language.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

It is often necessary to give detailed specifications of the object data representations to obtain maximum density for large data files to meet format requirements imposed by the hardware of peripheral equipment, to allow special optimizations on time critical paths, or to ensure compatibility when transferring data between machines.

It will be possible to specify the order of the fields, the width of fields, the presence of don't care fields, and the position of word boundaries. It will be possible to associate source language identifiers (data or program) with special machine addresses. The use of machine dependent characteristics of the object representation will be restricted as with machine dependent code (see J3). When multiple fields per word are specified the compiler may have to generate some form of shift and mask operations for source program references and assignments to those variables (i.e., fields). As with machine-language insertions, object data specifications should be used sparingly and the language features for their use must be Spartan, nor grandiose.

If the object representation of a composite data object is not specified in the source program, there will be no specific default guaranteed by the translator. The translator might, for example, attempt to minimize access time and/or memory space in determining the object representation. It might, depending on the object machine characteristics, assign variables and fields of records to full words, but assign array elements to the smallest of bits, bytes, half words, words or exact multiple words permitted by the logical description.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

The use of inline open procedures can reduce the run time execution costs significantly in some cases. There are the obvious advantages in eliminating the parameter passing, in avoiding the saving of return marks, and in not having to pass arguments to and from the routine. A less obvious, but often more important advantage in saving run time costs is the ability to execute constant portions of routines at compile time and, thereby, eliminate time and space for those portions of the procedure body at run time. Open routine capability is especially important for machine language insertions.

The distinction between open and closed implementation of a routine is an efficiency consideration and should not affect the function of the routine. Thus, an open routine will differ from a syntax macro in that (a) its global environment is that of its definition and not that of its call and (b) multiple occurrences of a formal value (i.e., read only) parameter in the body have the same value. If a routine is not specified as either open or closed the choice will be optimal as determined by the translator.

K. PROGRAM ENVIRONMENT
1. Operating System Not Required
2. Program Assembly
3. Software Development Tools
4. Translator Options
5. Assertions and Other Optional Specifications

K1. The language will not require that the object machine have an operating system. When the object machine does have an operating system or executive program, the hardware/operating system combination will be interpreted as defining an abstract machine which acts as the object machine for the translator.

A language definition cannot dictate the architecture of existing object machines whether defined entirely in hardware or in a hardware/software combination. It can provide a source language representation of all the needed capabilities and attempt to choose these so they have an obvious and efficient translation in the object machines.

K2. The language will support the integration of separately written modules into an operational program.

Separately written modules in the form of routines and type definitions are necessary for the management of large software efforts and for effective use of libraries. The user will be able to cause anything in any accessible library to be inserted into his program. This is a requirement for separate definition but not necessarily for separate compilation. The decision as to whether separately defined program modules are to be maintained in source or object language form is a question of implementation efficiency, will be a local management option and will not be imposed by the language definition. The trade-offs involved are complicated by other requirements for type checking of parameters (see C6), for open subroutines (see J5), for efficient object representations (see J1), and for constant expression evaluation at compile time (see C4). In general, separate compilation increases the difficulty and expense of the interface validations needed for program safety and reliability and detracts from object program efficiency by removing many of the optimizations otherwise possible at the interfaces, but at the same time it reduces the cost and complexity of compilation.

K3. A family of programming tools and aids in the form of support packages including linkers, loaders and debugging systems will be made available with the language and its translators. There will be a consistent easily used user interface for these tools.

The time has passed in which a programming language can be considered in isolation from its programming environment. The availability of programming tools which need not be developed and/or supported by individual projects is a major factor in the acceptability of a language. There is no need to restrict the kinds or form of support software available in the programming environment, and continued development of new tools should be encouraged and made available in a competitive market. It is, however, desirable that tools be developed in their own source language to simplify their portability and maintainability.

K4. A variety of useful options to aid generation, test, documentation and modification of programs will be provided as support software available with the language or as translator options. As a minimum these will include program editing, post- mortem analysis and diagnostics, program reformating for standard indentations, and cross-reference generation.

There will be special facilities to aid the generation, test, documentation and modification of programs. The "best" set of capabilities and their proper form is not currently known. Since nonstandard translator options and availability of nonstandard software tools and aids do not adversely affect software commonality, the language definition and standards will not dictate arbitrary choices. Instead, the development of language associated tools and aids will be encouraged within the constraint of implementing and supporting the source language as defined. Tools and debugging aids will be source language oriented.

Some of the translator options which have been suggested and may be useful include the following. Code might be compiled for assertions which would give run time warnings when the value of the assertion predicate is false. It might provide run time tracing of specified program variables. Dimensional analysis might be done on units of measure specifications. Special optimizations might be invoked. There might be capability for timing analysis and gathering run time statistics. There might be translator supplied feedback to provide management visibility regarding progress and conformity with local conventions. The user might be able to inhibit code generation. There might be facilities for compiling program patches and for controlling access to language features. The translator might provide a listing of the number of instructions generated against corresponding source inputs and/or an estimate of their execution times. It might provide a variety of listing options.

K5. The source language will permit inclusion of assertions, assumptions, axiomatic definitions of data types, debugging specifications;, and units of measures in programs. Because many assertional methods are not yet powerful enough for practical use, nor sufficiently well developed for standardization, they will have the status of comments.

There are many opinions on the desirability, usefulness, and proper form for each of these specifications. Better program documentation is needed and specifications of these kinds may help. Specifications also introduce the possibility of automated testing, run time verification of predicates, formal program proofs, and dimensional analysis. The language will not prohibit inclusion of these forms of specification if and when they become available for practical use in programs. Assertions, assumptions, axiomatic definitions and units of measure in source language programs should be enclosed in special brackets and should be treated as interpreted comments -- comments which are delimited by special comment brackets and which may be interpreted during translation or debugging to provide units analysis, verification of assertions and assumptions, etc.--but whose interpretation would be optional to translator implementations.

L. TRANSLATORS
1. No Superset Implementations
2. No Subset Implementations
3. Low-Cost Translation
4. Many Object Machines
5. Self-Hosting Not Required
6. Translator Checking Required
7. Diagnostic Messages
8. Translator Internal Structure
9. Self-Implementable Language

L1. No implementation of the language will contain source language features which are not defined in the language standard. Any interpretation of a language feature not explicitly permitted by the language definition will be forbidden.

This guarantees that use of programs and software subsystems will not be restricted to a particular site by virtue of using their unique version of the language. It also represents a commitment to freezing the source language, inhibiting innovations and growth in the form of the source language, and confining the base language to the current state of the art in return for stability, wider applicability of software tools, reusable software, greater software visibility, and increased payoff for tool building efforts. It does not, however, disallow library definition optimizations which are translator unique.

L2. Every translator for the language will implement the entire base language. There will be no subset implementations of the base language.

If individual compilers implement only a subset of the language, the there is no chance for software commonality. If a translator does not implement the entire language, it cannot give its users access to standard supported libraries or to application programs implemented on some other translator. Requiring that the full language be implemented will be expensive only if the base language is large, complex, and nonuniform. The intended source language product from this effort is a small simple uniform base language with the specialized features, support packages, and complex features relegated to library routines not requiring direct translator support. If simple low cost translators are not feasible for the selected language, then the language is too large and complex to be standardized and the goal of language commonality will not be achievable.

L3. The translator will minimize compile time costs. A goal of any translator for the language will be low cost translation.

Where practical and beneficial the user will have control over the level of optimization applied to his programs. The programmer will have control over the tradeoffs between compile time and run time costs. The desire for small efficient translators which can be hosted by machines with limited size and capability should influence the design of the base language against inclusion of unnecessary features and towards systematic treatment of features which are included. The goal will be effective use of the available machines both in object execution and translation and not maximal speed of translation.

Translation costs depend not only on the compiler but the language design. Both the translator and the language design will emphasize low cost translation, but in an environment of large and long-lived software products this will be secondary to requirements for reliability and maintainability. Language features will be chosen to ensure that they do not impose costs for unneeded generality and that needed capabilities can be translated into efficient object representations. This means that the inherent costs of specific language features is the context of the total language must be understood by the designers, implementers and users of the language. One consequence should be that trivial programs compile and run in trivial time. On the other hand, significant optimization is not expected from a minimal cost translator.

L4. Translators will be able to produce code for a variety of object machines. The machine independent parts of translators might be built independent of the code generators.

There is currently no common widely used computer in the DoD. There are at least 250 different models of commercial machines in use in DoD with many more specialized machines. A common language must be applicable to a wide variety of models and sizes of machines. Translators might be written so they can produce object code for several machines. This reduces the proliferation of translators and makes the full power of an existing translator available at the cost of producing an additional code generator.

L5. The translator need not be able to run on all the object machines. Self-hosting is not required, but is often desirable.

The DoD operational programming environment includes many small machines which are unable to support adequately the design, documentation, test, and

debugging aids necessary for the development of timely, reliable or efficient software. Large machine users should not be penalized for the restrictions of small machines when a common language is used. On the other hand, the size of machines which can host translators should be kept as small as possible by avoiding unnecessary generality in the language.

L6. The translator will do full syntax checking, will check all operations and parameters for type compatibility and will verify that all language imposed semantic restrictions on the source programs are met. It will not automatically correct errors detected at compile time.

The purpose of source language redundancy and avoidance of error prone language features is reliability. The price is paid in programmer inconvenience in having to specify his intent in greater detail. The payoff comes when the translator checks that the source program is internally consistent and adheres to its authors' stated intentions. There is a clear trade-off between error avoidance and programming ease; surveys conducted in the Services show that the programmers as well as managers will opt for error avoidance over ease when given the choice. The same choice is dictated by the need for well documented maintainable software.

L7. The translator will produce compile time explanatory diagnostic error and warning messages. A suggested set of error and warning situations will be provided as part of the language definition.

The translator will attempt to provide the maximal useful feedback to its user. Diagnostic messages will not be coded but will be explanatory and in source language terms. Translators will continue processing and checking after errors have been found but should be careful not to generate erroneous messages because of translator confusion. The translator will always produce correct code; when source programs errors are encountered by the translator or referenced program structures omitted, the compiler will produce code to cause a run time exception condition upon any attempt to execute those parts of the program. Warnings will be generated when a source language construct is exceptionally expensive to implement on the specified object machine. A suggested set of diagnostic messages provided as part of the language definition contributes to commonality in the implementation and use of the language. The discipline of designing diagnostic messages keyed to the design may also uncover pitfalls in the language design and thereby contribute to a more precise and better understood language description.

L8. The characteristics of translator implementations will not be dictated by the language definition or standards.

The adoption of a common language is a commitment to the current state of the art for programming language design for some duration. It does not, however, prevent access to new software and hardware technology, new techniques and new management strategies which do not impact the source language definition. In particular, innovation should be encouraged in the development of translators for a common language providing they implement exactly the source language as defined. Translators like all computer programs should be written in expectation of change.

L9. Translators for the language will be written in their own source language.

There will be at least one implementation of the translator in its own language which does all parsing and compile-time checking and produces an output suitable for easy translation to specific object machines. If the language is well-defined and uniform in structure, a self-description will contribute to understanding of the language. The availability of the machine independent portion of a translator will make the full power of the language available to any object machine at the cost of producing an additional code generator (whose cost may be high) and it reduces the likelihood of incompatible implementations. Translators written in their own source language are automatically available on any of their object machines providing the object machine has sufficient resources to support a compiler.

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL
1. Existing Language Features Only
2. Unambiguous Definition
3. Language Documentation Required
4. Control Agent Required
5. Support Agent Required
6. Library Standards and Support Required

M1. The language will be composed from features which are within the state of the art and any design or redesign which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project.

The adoption of a common language can be successful only if it makes available a modern programming language compatible with the latest software technology and is compatible with "best" current programming practice but the design and implementation of the language should not require additional research or require use of untried ideas. State-of-the-art cannot, however, be taken to mean that a feature has been incorporated in an operational DoD language and used for an extended period, or DoD will be forever tied to the technology of FORTRAN-like languages; but there must be some assurances through analysis and use that its benefits and deficiencies are known. The larger and more complex the structure, the more analysis and use that should be required. Language design should parallel other engineering design efforts in that it is a task of consolidation and not innovation. The language designer should be familiar with the many choices in semantic and syntactic features of language and should strive to compose the best of these into a consistent structure congruous with the needed characteristics. The language should be composed from known semantic features and familiar notations, but the use of proven feature should not necessarily impose that notation. The language must not just be a combination of existing features which satisfy the individual requirements but must be held together by a consistent and uniform structure which acts to minimize the number of concepts, consolidates divergent features and simplifies the whole.

M2. The semantics of the language will be defined unambiguously and clearly. To the extent a formal definition assists in attaining these objectives, the language's semantics will be specified formally.

A complete and unambiguous definition of a common language is essential. Otherwise each translator will resolve the ambiguities and fill in the gaps in its own

unique way. There are currently a variety of methods for formal specification of programming language semantics but it remains a major effort to produce a rigorous formal description and the resulting products are of questionable practical value. The real value in attempting a formal definition is that it reveals incomplete and ambiguous specifications. An attempt will be made to provide a formal definition of any language selected but success in that effort should not be requisite to its selection. Formal specification of the language might take the form of an axiomatic definition, use of the Vienna Definition Language, or use of some other formal semantic system.

M3. The user documentation of the language will be complete and will include both a tutorial introductory description and a formal in-depth description. The language will be defined as if it were the machine level language of an abstract digital computer.

The language should be intuitively correct and easily learned and understood by its potential users. The language definition might include an Algol-60 like description(P. Naur (Ed.), "Revised Report on the Algorithmic Language Algol-60," Communication of the A.C.M. Vol.6, No. 1, January 1963, p. 1-17.) with the source language syntax given in BNF or some other easily understood metalanguage and the corresponding semantics given in English. As with the descriptions of digital computer hardware, the semantics and syntax of each feature must be defined precisely and unambiguously. The action of any legal program will be determinable from the program and the language description alone. Any computation which can be described in the language will ultimately draw only on capabilities which are built into the language. No characteristics of the source language will be dependent on the idiosyncrasies of its translators.

The language documentation will include syntax, semantics and examples of each language construct, listings of all key words and language defined defaults. Examples shall be included to show the intended use of language features and to illustrate proper use of the language. Particularly expensive and inexpensive constructs will be pointed out. Each document will identify its purpose and prerequisites for its use.

M4. The language will be configuration managed throughout its total life cycle and will be controlled at the DoD level to ensure that there is only one version of the source language and that all translators conform to that standard.

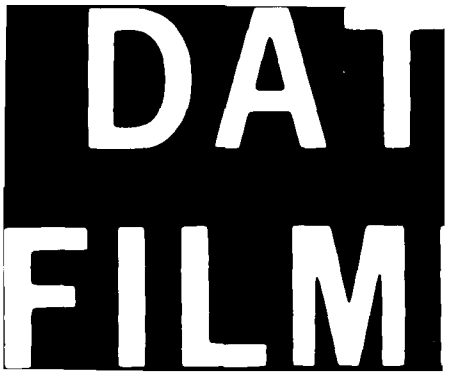
Without controls a hopefully common language may become another umbrella under which new languages proliferate while retaining the same name. All compilers will be tested and certified for conformity to the standard specification and freedom from known errors prior to their release for use in production projects. The language manager will be on the OSD staff, but a group within the Military Departments or Agencies might act as the executive agent. A configuration control board will be instituted with user representation and chaired by a member of the OSD staff.

M5. There will be identified support agent(s) responsible for maintaining the translators and for associated design, development, debugging and maintenance aids.

Language commonality is an essential step in achieving software commonality, but the real benefits accrue when projects and contractors can draw on existing software with assurance that it will be supported, when systems can build from off the shelf components or at least with common tools, and when efforts can be spent to expand existing capabilities rather than building from scratch. Support of common widely used tools and aids should be provided independent of projects if common software is to be widely used. Support should be on a DoD-wide basis with final responsibility resting with a stable group or groups of qualified in-house personnel.

M6. There will be standards and support agents for common libraries including application-oriented libraries.

In a given application of a programming language three levels of the system must be learned and used: the base language, the standard library definitions used in that application area, and the local application programs. Users are responsible for the local application programs and local definitions but not for the language and its libraries which are used by many projects and sites. A principal user might act as agent for an entire application area.


DAT
FILM